Computer Science

COMPSCI 105 S2 C - Assignment 3

Due Date: Friday 19 October 2012 at 4:30pm

75 marks in total = 7.5% of the final grade

Assessment

- Due: 19 October, 2012 (4:30 pm)
- Worth: 7.5% of your final mark

Files

- A copy of this handout, as well as relevant files for each of the questions in this assignment can be obtained from the 105 assignments page on the web:

  http://www.cs.auckland.ac.nz/compsci105s2c/assignments/

- Using the web drop box (https://adb.ec.auckland.ac.nz/adb/), you may submit the following file for this assignment:

  - Question: `TextZip.java`

Aims of the assignment

- solving problems using Binary Trees
- recursive algorithms, file I/O

Warning

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work, modify it, and submit it as your own.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never loan your code to another person.
  - Never put your code in a public place (e.g., forum, your web site).
  - Never leave your PC without locking the screen (e.g., to get food, to have a drink, or to go to the toilet). You are responsible for the security of your account.
  - Never get code from a tutor (e.g., private tutors). Several tutors have been caught giving the same code to all their students.
  - Always reference the source for text you copy as part of the answer to an assignment.

Your name, UPI and other notes

- In *all questions* your UPI **must** appear somewhere in the output for the program.
- In this handout, the UPI `abcd001` has been used to illustrate this in the examples, however make sure that for the programs **you** submit, **your** UPI is included somewhere in the output.
- All files should also include your name and ID in a comment at the beginning of the file.
- All your files should be able to be compiled without requiring any editing.
- All your files should include adequate documentation - note that you are not required to produce full Javadoc comments.
- This assignment has a template code frame (`TextZip.java`) that you can work on. And it also has a set of resource files (`A3Resource.zip`) that you may find useful for the assignment. You do not have to use all of the resource files, but you must NOT modify any of them, because the markers will use the unmodified versions of the resource files to mark your assignment. The resource files are NOT to be submitted.

Question *(75 Marks)*

In this question you need to implement a basic data compression / decompression program using the popular Huffman compression algorithm on Binary Trees.

We know that characters in a text file are often represented as ASCII code in a computer. ASCII code is a fixed length coding that uses patterns of seven binary digits to represent each character. Nowadays, many computers use eight-bit bytes, where the eighth bit was commonly used as a parity bit for error checking. Machines that do not use parity checking typically set the eighth bit to '0'. For example, a text file that consists of the characters "abracadabra!" can be represented in the following 12 bytes of ASCII coding.

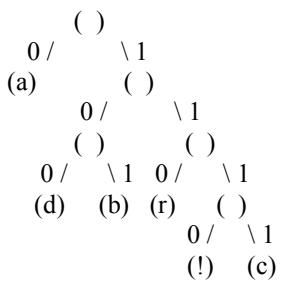
```
   a        b        r        a        c        a        d        a        b
01100001 01100010 01110010 01100001 01100011 01100001 01100100 01100001 01100010

   r        a        !
01110010 01100001 00100001
```

ASCII is an example of a fixed length code. The ASCII character set treats each character in the alphabet equally, and makes no assumptions about the frequency with which each character occurs. The basic idea behind this text compression program is to use a variable length coding technique that assigns a shorter code representation to frequently used character in the text. By doing so, the total length of the character representations of the text (file size in bytes) can be reduced dramatically.

A variable length code is based on the idea that for a given alphabet, some letters occur more frequently than others. The Huffman algorithm produces an optimal variable length prefix code for a given alphabet in which frequencies are pre-assigned to each letter in the alphabet. Symbols that occur more frequently have a shorter code length than symbols that occur less frequently. The two symbols that occur least frequently will have the same code length.

A prefix code is most easily represented by a Binary Tree, in which the external (leaf) nodes are labeled with single characters that are combined to form the message. The encoding for a character is determined by following the path down from the root of the tree to the external node that holds that character, i.e., a '0' bit identifies a left branch in the path, and a '1' bit identifies a right branch. In the following Binary Tree, the code for letter 'c' is "1111", because the external node holding 'c' is reached from the root by taking 4 consecutive right branches. An example of prefix codes is given in the table below.



Sample character encoding

!	1110
a	0
b	101
c	1111
d	100
r	110

In the example above, the character 'a' is encoded with a single bit '0', while the character 'c' is encoded with 4 bits. This is a fundamental property of prefix codes. In order for this encoding scheme to reduce the number of bits in a message, we use short encodings for frequently used characters, and long encodings for infrequent ones. A second fundamental property of prefix codes is that the coding for each character is unique, which means messages can be formed by simply stringing together the code bits from left to right. For example, using the above prefix tree, the following bit string,

0101110011110100010111001110

encodes the message "abracadabra!". The first 0 must encode 'a', then the next three "101" must encode 'b', then "110" must encode 'r', and so on as follows.

```
|0|101|110|0|1111|0|100|0|101|110|0|1110
 a b  r a c  a d a b  r a  !
```

The codes can be run together because no encoding is a prefix of another one. This property defines a prefix code, and it allows us to represent the character encodings with a binary tree and decode them uniquely as shown above.

Algorithms and tasks

For this question, your program `TextZip` should implement the following six tasks defined in four sub-questions.

(1) Given a default prefix code tree and its compressed file ("a.txz"), decompress the file and save the uncompressed file into a text file ("a.txt"), also output sizes of the files and compression ratio;

Possible algorithm:

- Use the `BitReader` class to read each bit of the compressed file ("a.txz") and decompress it. The algorithm for decoding characters is as follows.
  - Start at the root of the prefix tree.
  - Repeat until you reach an external leaf node.
    - Read one message bit.
    - Take the left branch in the tree if the bit is '0'; take the right branch if it is '1'.
  - Read the character in that leaf node and write it into the output text file.
  - Repeat the above process until reach the end of the compressed file.
- Read the file size of both the compressed and decompressed files, calculate the compression ratio = (compressed file size / uncompressed file size)

Your program should have similar input/outputs as follows:

```
c:\105S1T\A3> java TextZip –a
a.txz decompressed by abdc001
Size of the compressed file: 5 bytes
Size of the original file: 12 bytes
Compressed ratio: 41.66666666666667%
```

As from the above result, the original message contains 12 bytes which would normally require 96 bits of storage (8 bits per character). The compressed message uses only 5 bytes, or 41.67% of the space required without compression. The compression factor depends on the frequency of characters in the message, but ratios around 50% are common for English text. Note that for large messages the amount of space needed to store the description of the tree is negligible compared to storing the message itself, so we have ignored this quantity in the calculation.

Note that you also need to save the decompressed file into a text file ("a.txt"). You should implement the `decompress` method in the code frame given with the following method header.

```
public static void decompress(BitReader br,
                   TreeNode<CharFreq> huffman, FileWriter fw)
```

Where `BitReader` is a file reader that will read the compressed file bit by bit; `huffman` is the root node of the prefix code tree; `FileWriter` is a standard Java class for output to files. Note that the data object in the `TreeNode` is an instance of the `CharFreq` class (definition already provided in "`A3Resource.zip`").

(2) Given a text file (e.g., "filename.txt"), calculate the frequencies of each characters in the file and save the character frequency table into a file (e.g., "filename.freq"); based on the character

frequencies, build up the prefix code tree using Huffman algorithm and print out the prefix codes for each characters appeared the text file.

(2.1) calculate the character frequencies in a text file

Possible algorithm:

- Using an array of integers (`int []`) to store the frequency for each character

char			a	b		e		s	t	
frequency	0	…	10	0	…	15	…	3	4	…
index	0		97	98		101		115	116	

when you read a character from the file, lookup the corresponding indexed (ASCII code) items in the array and increment its frequency.

- Save the non-zero frequencies and their character into a file (e.g., "filename.freq")
- Create and return an array list of tree node from the non-zero frequencies characters in the frequency array. Note that the tree node list is originally arranged according to their ASCII order from left to right.

In this sub-question you should implement the `countFrequencies` method in the code frame given with the following method header.

```
public static ArrayList<TreeNode<CharFreq>> countFrequencies
                                (FileReader fr, PrintWriter pw)
```

where `FileReader` is a standard Java class to read in files, and `PrintWriter` is a standard Java class to output to files. This method return an `ArrayList` of tree node, where the data object in the `TreeNode` is an instance of the `CharFreq` class (definition provided in "`A3Resource.zip`").

(2.2) build up the prefix code tree using Huffman algorithm

Possible algorithm:

- The Huffman algorithm works by constructing a prefix code tree from the bottom up, using the frequency counts of the symbols to repeatedly merge subtrees together. Intuitively, the symbols that are more frequently occurred should appear higher in the tree structure and the symbols that are less frequent should be lower in the tree. Let's use an example character table to illustrate this:

Character	Frequency	Sum (bits)
!	1	8
a	5	40
b	2	16
c	1	8
d	1	8
r	2	16
Total		96

Now we simply create a "forest" of six (one for each character) binary trees. This should be obtained by passing an array list of tree nodes into the method call.

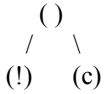
1	5	2	1	1	2
(!)	(a)	(b)	(c)	(d)	(r)

Weight of the tree (here: frequency)

Next we remove the least frequency element (tree) in the list from the left to right twice. Then merge these two minimum-weight trees, where the latest extracted element will be the right subtree. The new merged tree will have a weight of sum of its two subtrees, append the merged tree onto the end of the tree node list.

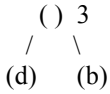
The sum of the weights: 2

5	2	1	2	2
(a)	(b)	(d)	(r)	( )



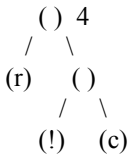
In the next step we again merge the trees of minimum weight:

5	2	2	3
(a)	(r)	( )	( )



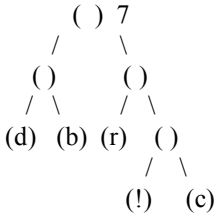
Step 3:

5	3	4
(a)	( )	( )



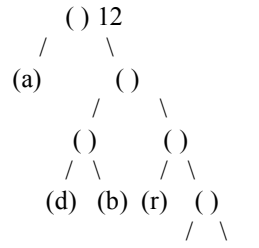
Step 4:

5	7
(a)	( )



Final step:

12
( )



Thus our optimal prefix code using Huffman algorithm is as follows.

Character	Code	Frequency	Sum (bits)
!	1110	1	4
a	0	5	5
b	101	2	6
c	1111	1	4
d	100	1	3
r	110	2	6
Total			28

In this sub-question you should implement the `buildTree` method in the code frame given with the following method header.

```
public static TreeNode<CharFreq>
        buildTree(ArrayList<TreeNode<CharFreq>> trees)
```

where `trees` is an `ArrayList` of tree node, where the data object in the `TreeNode` is an instance of the `CharFreq` class containing a character and its frequency. This method returns the root of the prefix code tree generated using Huffman algorithm.

(2.3) print out the prefix code for each character from prefix codes tree generated.

Possible algorithm

- Modify the common tree traversal method, traverse the prefix code tree and print out the prefix codes. The following recursive algorithm might be useful.
  - If it is a leaf node, print out the character and its prefix code

- Otherwise go to the left subtree and keep track of '0'; go to the right subtree and keep track of '1'

In this sub-question you should implement the `traverse` method in the code frame given with the following method header.

```
public static void traverse(TreeNode<CharFreq> t,String code)
```

where `t` is a tree node in the prefix code tree, `code` is a string that keeps the track of the route taken.

Your program should have similar input/outputs as follows:

```
c:\105S1T\A3> java TextZip –f a.txt a.freq
a.txt prefix codes by abdc001
character code:
a : 0
d : 100
b : 101
r : 110
! : 1110
c : 1111
```

Note that you also need to save the character frequency into a file (e.g., "a.freq"), which has the format as follows.

```
! 1
a 5
b 2
c 1
d 1
r 2
```

(3) Given a text file (e.g., "filename.txt"), create the prefix code tree using the Huffman algorithm and save its frequency table into a file (e.g., "filename.freq"); using the prefix codes generated to compress the text file and save the compressed content into a file (e.g., "filename.txz"), also output the file sizes and compression ratio.

Possible algorithm:

- Reuse the methods implemented in previous questions (i.e., `countFrequency`, `BuildTree`) to generate the prefix code tree using Huffman algorithm;
- Redefine a similar `traverse` method that implemented in the previous question to store the character and its prefix code in to a table structure for future lookup purpose;
- Use the prefix code table to lookup the code for each character in the text file, perform the compression by replacing each character with its prefix code, use the `BitWrite` class to write each bit into a compressed file (e.g., "filename.txz");
- Since you will be using the `countFrequency` method, it will also save the character frequency table into a file (e.g., "filename.freq") for future decompression usage.

Your program should have similar input/outputs as follows:

```
c:\105S1T\A3> java TextZip -c file.txt file.txz file.freq
file.txt compressed by abdc001
Size of the compressed file: 932302 bytes
Size of the original file: 1749990 bytes
Compressed ratio: 53.27470442688244%
```

- Note that the file '`file.txt`' used in the above example has the exact same content as the file '`dictionary_large.txt`' from the '`A3Resource.zip`'.

(4) Given a compressed file (e.g., "filename.txz") and its frequency file (e.g., "filename.freq"), create the prefix code tree using the Huffman algorithm. Then use the prefix codes tree to decompress the "file.txz" file and save the uncompressed content into a text file (e.g., "filename.txt"), also output the file sizes and compression ratio.

Possible algorithm:

- Reuse the methods implemented in previous questions (e.g., `countFrequencies` or `BuildTree`) to generate the prefix code tree using Huffman algorithm. You need to redefine a similar `readFrequencies` method in order to generate frequency array list from the saved frequency table file (e.g., "filename.freq") instead of counting them from a text file;
- Once you correctly obtained the frequency array list from the frequency table, you can reuse the `BuildeTree` and `decompress` methods implemented in the previous questions to perform the decompression and save the result back to a text file (e.g., "filename.txt").

Your program should have similar input/output as follows:

```
c:\105S1T\A3> java TextZip -d file.txz file.freq file.txt
file.txz decompressed by abdc001
Size of the compressed file: 932302 bytes
Size of the original file: 1749990 bytes
Compressed ratio: 53.27470442688244%
```

NOTE:

- Your UPI must appear in the output, as shown in the example above

FEEDBACK FILE for the assignment

- A3.txt - a text file containing your feedback on the assignment (see below). You must include a text file named A3.txt in your submission. There will be a 5 marks penalty for not doing so. This text file must contain the following information:

  Your full name
  Your login name and ID number
  How much time did the assignment take overall?
  What areas of the assignment did you find easy?
  What areas of the assignment did you find difficult?
  Any other comments you would like to make.

Marking details

Your UPI must appear in the output for every question. If it does not appear in the output, you will forfeit the marks for that question. All files should include your name and ID in a comment at the beginning of the file.

Question	***75 Marks***
Submit a file named `TextZip.java`, and must use a Binary Tree data structure to solve the problem.	
Documentation/ Neatness/ Style	4
Correctly decompress the default "a.txz" file using the default prefix code tree provided, output the compression ratio and the "a.txt" file	15
Correctly count the character frequency from the input text file, output the frequency table to a "*.freq" file	10
Generates the prefix code tree using the Huffman algorithm	12
Correctly traverse the prefix code tree and display the character prefix codes	10
Correctly compress any text file, save the compressed text file into a "*.txz" file, and compute the compression ratio, and also save the character frequencies of the text file into a "*.freq" file	12
Correctly decompress any compressed file in "*.txz" format with its frequency file "*.freq" into its original text format file "*.txt" , and compute the compression ratio	12